From stack removing in stack-based languages to BibTEX++

Emmanuel Donin de Rosière
Supervisor : Ronan Keryell

The 28th August 2003

MASTER THESIS
2002-2003

ENST Bretagne
Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex

Dedicated to Safaa

RÉSUMÉ

Dans ce rapport, nous présentons *BibTEX++*, un logiciel de gestion de références bibliographiques qui essaie de devenir le successeur du très connu *BibTEX*. L'un des fonctionnalités les plus interessantes de *BibTEX++* est la possibilité de tranformer les fichiers de style pour BibTEX (écrits en *bst*) en fichier de style pour BibTEX++ (qui sont écrits en JAVA). Nous allons donc expliquer comment nous avons transformé la pile du language bst en variables JAVA, comment nous avons retrouvé le type de ces variables et quelles optimizations nous avons utilisées afin de rendre les fichiers de style compilés les plus compréhensibles possible.

Dans une première partie, nous allons donc présenter quelques concepts importants, puis expliquer le fonctionnementde *BibTEX++* et *BiSTrO*, le compilateur *bst→JAVA* et enfin exposer les différents résultats obtenus.

Mots clefs : BibTEX, langage à pile, optimisations, compilateur, BibTEX++, bibliographie, LATEX

ABSTRACT

In this report, we introduce *BibTEX++*, a bibliography section creator for LaTeX which tries to become the *BibTEX* successor. One of the main features of this software is to compile old *BibTEX* style files (*bst* language) to JAVA code. So, we will explain how we transform the stack of the bst language into JAVA variables, how we find the type of each element on the stack and what optimizations we use in order to clarify the output code.

First, we are going to present some important concepts and then explain how works *BibTEX++* and *BiSTrO* (the *bst*→*JAVA* compiler) and finally show you the results of these softwares.

Keywords : BibTeX, stack-based language, optimizations, compiler, BibTeX++, biliography, LaTeX

ACKNOWLEDGMENTS

First, I would like to thank all students and researchers which worked on this project : Laura Barrero, Martin Brisbarre, Laurent Cordival, Fabien Dagnat, Etienne Debenoist, Guillaume Ferrier, Aude Jacquot, Olivier Muller, Mathieu Servillat, Firass Squalli, Nicolas Torneri and Emmanuel Valliet.

I also want to express my gratitude to Ronan Keryell for his valuable help and technical support. He was (almost ?!) always there to listen and help.

Table of contents

Résumé **iv**

Abstract **v**

Aknowledgments **vii**

Table of contents **x**

1 Introduction **1**

2 Background **3**
2.1 BibTEX . . . 3
2.2 Stack-Based Language . . . 3
2.3 BST language . . . 5
2.4 Stack removing in stack-based language . . . 8
2.4.1 Source-to-source translator . . . 8
2.4.2 Compiler . . . 12
2.4.2.1 RAFTS . . . 12
2.4.2.2 LaTTe . . . 13

3 BibTEX++ **15**
3.1 Presentation of BibTEX++ . . . 16
3.2 BiSTrO architecture . . . 17
3.2.1 Description . . . 17
3.2.2 The Parser / Lexer . . . 18
3.2.3 The Tree translator . . . 21
3.2.4 The Optimizer . . . 21
3.3 Translation . . . 22
3.3.1 Stack representation . . . 22

3.3.2 Built-in functions 23
3.3.3 Type inference on stack item 25
3.3.4 Return parameters 27
3.3.5 Control structures 27
3.3.6 Unknown stack depth 27
3.4 Optimizations 28
3.4.1 Copy propagation 28
3.4.2 Constant propagation 29
3.4.3 Dead code elimination 29
3.5 Security 30

4 Results **31**
4.1 Speed and execution time 31
4.2 BibTEX++ styles and optimizations 33

Conclusion **37**

Appendices **41**

A *bst* grammar **41**

B Error in `format.names` **47**

Bibliography **50**

Chapter 1

Introduction

A bibliography or list of references is a listing of all sources from which you have taken information directly (by literal quotation) or indirectly (through paraphrase), or where you have used information or reproduced material from. These references are used to :

- Clearly identifying each document. So it provides some identification elements to allow the reader to search these documents in librairy catalogues or anywhere else. In most cases, these identification elements are normalized [10] (we often use the name of the book, the author, the editor, ...), but because of the numerical revolution, there are more and more document types (web pages for example) and identification elements (e-mail, URL . . .). So the management of the references became more and more difficult.

- Enable the reader to consult the sources you have used with a minimum of effort. So we have to indicate precisely where (on which page, the electronic location) or under which circumstances (personal interview, e-mail) you obtained the information.

But, creating a bibliography has always been an headache for typographers : if the article said [12] is a book about meta-middleware and in the bibliography [12] is a cook book, there has been a problem somewhere. Another problem is each journal has its own bibliographic style, so a bibliographic management software has to allow the user to create his own style and send it to other people.

Today, everybody mostly use only one bibliographic software for LaTeX : it's *BibTeX* [16]. But it hasn't changed for ten years, so it can't use modern

encoding (such as unicode UTF-8) and the BibTEX style language is not very expressive. Today, there are only few people that are able to create new styles from scratch. It's in this context that the project *BibTEX++* takes place. It tries to be his successor with more features but fully compatible with *BibTEX*.

In order to continue this project, I realized a master thesis at the computer science department of the *ENST Bretagne* (the National Telecommunication Engineering School of Brittany) in France. My supervisor in this project was Ronan KERYELL, an assistant professor at the Laboratory of Computer Science and Telecommunications (LIT) in this school.

Chapter 2

Background

2.1 BibTEX

BibTEX is a popular tool used by the LaTeX [8] community to generate bibliographical notices in publications. It suites very well the LaTeX philosophy and is well integrated to it : the user describes his wishes with a mark-up language but not the layout details : the LaTeX infrastructure will use some styles to typeset the presentation from the high level content description.

The citation matter is stored in a database (a file with a *.bib* extension) and *BibTEX* picks from the *.aux* file the needed information according to the citation marks placed in the LaTeX document (*.tex* file) by the author. A typeset version of the bibliography is done by *bibTEX* to the *.bbl* file by using a *.bst* bibliography style file. The work-flow is summed up on figure 2.1 [8].

2.2 Stack-Based Language

BibTeX *bst* files use a stack-based language : it is a type of language where you have to put data you want to use in a stack. Functions store their results in this stack (in fact we use an inversed polish notation -IPN- or postfix notation). For example, if you want to do 2 + 3, you have to put 2 and 3 on the stack and then call the add operor, so you have to write something like : `2 3 +`.

Stack-based languages are sometimes used as programming languages but they are more popular as intermediate languages for compiler and as machine-independent executable program representations. Here is a list of stack-based

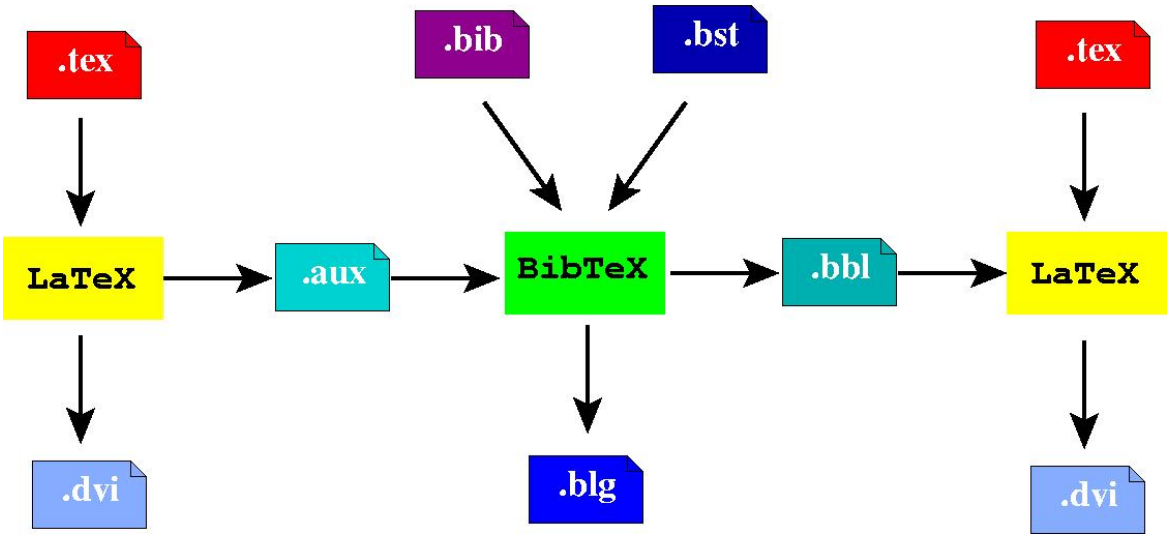

Figure 2.1: *BibTEX* Work-flow

languages :

Programming languages

- **Forth** is used in many fields, especially in embedded control applications.
- **PostScript** is used primarily in typesetting and other display purposes. But because the majority of PostScript code is written by programs, it can also be regarded as an intermediate language.
- **RPL** (reverse polish LISP) is the language of the HP-48 calculator; It's a run-time type-checked language with mathematical data types (it's on a calculator) and Forth-like syntax.
- **The *BibTEX* style language** or *bst* language is a stack-based language for processing *BibTEX* databases.

Itermediate language

- **UCSD p-code** is used in the UCSD system, an operating system well-known for its Pascal compiler. P-code was either interpreted or compiled to native code. We can find today p-code compilers for more recent system.

- **Smalltalk-80 byte code** is the intermediate language of the Smalltalk-80 system.

Machine-independent language

- **JAVA bytecode** is the code you obtain after compiling a JAVA file. It can be used on every system supported by JAVA because it will be interpreted by the JAVA Virtual Machine.

But computers are mainly register machines, so it's not easy to implement a stack-based language. That is why stack-based language are not frequently used. Today, there are 3 ways to execute a stack-based language on a register machines :

- Using an interpreter. At the execution, it will transform each instruction into a mnemonic one. The execution is dynamic but very slow.
- Using a compiler. It will statically transform the code into mnemonic one.
- Using a *source-to-source translator* which converts stack-based code into a high-level language and then compiling the output code. It allows the code to be executed in lots of different systems if the high-level language is well-known and used.

2.3 BST language

We have already seen that the *bst* language is the Style language for *BibTEX*. But it uses a specifical syntax that we will see here. A style file is program which formats the reference list in a certain way. For example, a style file can sort the reference list by alphabetical order thanks to the autor names and makes the title in italics. *BST* language uses ten commands which manipulate language objects : constants, variables, functions, the stack and the entry list (the reference list). A string value is between double-quotes like `"abcd efgh"` and an integer is preceded by an `#` like `#23`. There are also three different types of variables :

- Global variables which are declared by an `INTEGERS` or `STRINGS` comand.

- Entry variables which can be strings or integers, but a value is assigned for each entry of the list.
- Fields which are read-only strings. They represent information from the curent reference item, so each one has a value for every entry.

These ten commands are :

- `ENTRY` which declares the fields (in the bibliography databases) and the entry variables. `crossref` is a field which is automatically declared (used for cross referencing) and `sort.key$` is an entry variable (used for sorting references) also automatically declared.
- `EXECUTE` which executes a function. It takes one argument : the function name.
- `FUNCTION` which declares a new function. It takes two arguments : the function name and its definition. You have to declare a function before using it. So there is no recursion inside a bst file.
- `INTEGERS` which declares new integer global variables. `entry.max$` and `global.max$` are automatically declared (used for limiting the size of the strings).
- `ITERATE` which executes a single function for each entry in the reference list. These calls are made in the list current order.
- `MACRO` which defines a string macro. The first argument is the name of the macro and the second is the text remplacement. Each occurrence of the name of the macro will be replaced by the second argument. It's usefull for month abbreviation for example.
- `READ` which reads the database file and assigns to fields their value for each entry.
- `REVERSE` which performs the same action than `ITERATE` but in reverse order.
- `SORT` which sorts the reference list in alphabetical order according to `sort.key$`.
- `STRINGS` which declares new string variables.

All these commands permit to define the structure of a style file. But with them, we can not manipulate variables. It is why some built-in functions have been declared in *BibTEX*. There are 37 functions so we won't see each of them, but only the most important. If you want to know more, read *BibTEXing* written by Oren Patashnik [16]. The most important built-in function are :

<, <, =, + and - which manipulate integer variables and constants. In the *BST* language there is no boolean, so we use 1 when something is true and 0 otherwise.

`*` which concatenates two strings.

`:=` which assigns to the first literal the value of the second.

`call.type$` which executes the function whose name is the entry type of the entry. For example, it calls the function `book` if the entry is a book... It is often associated with the `ITERATE` command.

`format.name$` which pops the three tops literals from the stack. The last literal is a name list, so this function will format a name in this list with the specification of the first string.

`if$` which pops three literals. If the last is greater than 0, the second literal is executed else the first.

`skip$` which does nothing (usefull for creating `if$` empty branch).

`while$` which pops two literals and keeps executing the second as long as the first literal puts on the stack is greater than 0.

`write$` which writes the top string item into the bbl file (the output of *BibTEX*).

With all these build-in functions, we can create new function a bit more sophisticated like this one :

```
FUNCTION {and}
{   'skip$
    { pop$ #0 }
  if$
}
```

This function calculates the "logical and" between two numbers : if the first element on the stack is greater than 0, `skip$` is executed so this function returns the second element on the stack. Else, `pop$ #0` is executed which puts 0 on the stack. We can see even with this oversimplistic example that it is not easy to understand BST language :

- ➥ We are not accustomed with this postfix notation.
- ➥ The number and the type of input and output variables are implicit.
- ➥ We have to read all the control structure in reverse order.

It's why only few people are able to program a new style in this language. So for *BibTEX++*, we will have to create a style language more expresssive. But, because of the need of compatiblity with *BibTEX*, we'll have to transform this stack-based language into a standard one. So we will see how to remove the stack in a stack-based language.

2.4 Stack removing in stack-based language

2.4.1 Source-to-source translator

There are only few source to source translators for stack-based language. The most famous research on this was made by M. Anton Ertl (the author of *A New Approach to Forth Native Code Generation* [6]). During his PhD Thesis [5], he tried to transform Forth code into C language in order to increase the portability of Forth applications. Actually, if you want to use a Forth application on a special system, you have to develop a special interpreter. However, if you transform the Forth into C, it must be used on every system (because there is often a C compiler available). And you don't even need to optimize the C code because the C compiler will do that !

Practically all the other source to source translators for stack-based language are JAVA decompilers like : *krakatoa* [17] or *mocha* [18] which try to transform JAVA bytecode into Java. With this, you can obtain the sources of a program from compiled files. Nevertheless, all these translators use the same algorithm and special optimizations for JAVA and JAVA Bytecode. So we will only look at the M. Anton Erl one (this translator is called *f2c*).

f2c uses several steps to convert Forth code to C language. The first one is to split the code into its functions which will be processed independently.

Then, *f2c* counts for each function the number of input and output parameters. The next step is to convert the elements on the stack into C's local variables with the notation of the figure 2.2.

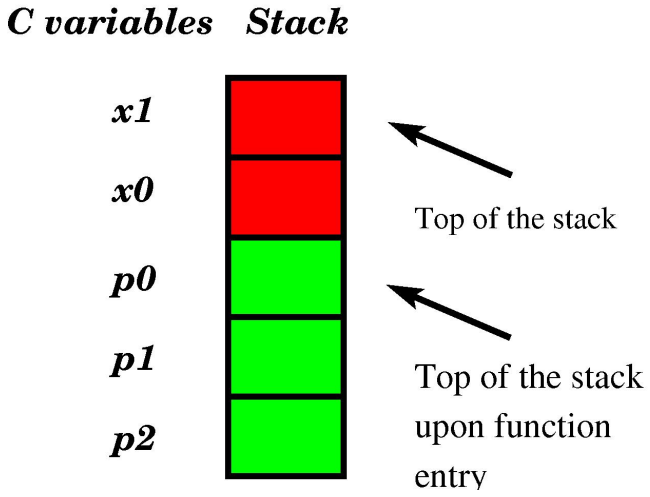

Figure 2.2: Relation between C variables and the stack.

So, `p0`, `p1`... represent the entry variables of each function and `x0`, `x1`... are used like local variables. This scheme ensures that stack items that are not affected by an operation do not have to be copied around between local variables.

Then *f2c* converts each Forth primitive into a C's sequence. For exemple, if the top of the stack resides in `x1`, the translation of `+` will look like :

```
{
  Cell n1=x1;
  Cell n2=x0;
  Cell n;
  n = n1+n2;
  x0=n;
}
/* top of the stack now: x0 */
```

This sequence is very long, but a good C compiler can compile it to only one instruction (sometimes, it can convert severals sequences into one instruction). So the translation always works like this :

- ➥ All the usefull elements are declared as C's local variables and are initialized.
- ➥ We copy the C code for the Forth primitive.
- ➥ The result variables are copied back to the stack.

f2c has also to convert all the control structures. But Forth allows the creation of arbitrary control structures so it is easier to convert them into `goto`s and `label`s C instructions.

Forth control Word	Meaning	C traduction
`IF`	conditional forward branch	`if (`*x*`==0) goto` *label* `;`
`AHEAD`	unconditional forward branch	`goto` *label* `;`
`THEN`	target of forward branch	*label* `:`
`BEGIN`	target of backward branch	*label* `:`
`UNTIL`	conditional backward branch	`if (`*x*`==0) goto` *label* `;`
`AGAIN`	unconditional backward branch	`goto` *label* `;`

Table 2.1: Forth control structures and their C traduction

Nevertheless, this translation mechanism needs to know the height of the stack everywhere in the Forth code, but it's not always possible. Sometimes the stack depth is unknown. For exemple, in the case of an instruction like `?DUP 0= IF` which means that if the top of the stack is `0`, we remplace it with the previous element on the stack, else we delete the element. So in this case, *f2c* has to create a C stack and uses it in the whole function.

f2c can easily convert a Forth program into a C one, but the output code is often not very expressive : for exemple the next Forth function which calculates the maximum between two numbers

```
2dup < if
    swap drop
else
    drop
endif
```

will be converted by *f2c* into :

```
Cell max(Cell p0, Cell p1)
{
/* stack now: ... p1 p0 */
{   /* 2dup */
    Cell n1,n2;
    n1=p0;
    n2=p1;
    p1=n2;
    p0=n1;
    x0=n2;
    x1=n1;
}
/* stack now: ... p0 x0 x1 */
{   /* less than */
    Cell n1,n2,n;
    n1=x1;
    n=FLAG(n2<n1);
    xo=n;
}
/* stack now: ... p0 x0 */
if (!x0) goto label0;
/* stack now: ... p1 p0 */
{   /* swap */
    Cell n1,n2;
    n1=p0;
    n2=p1;
    p1=n1;
    p0=n2;
}
/* stack now: ... p1 p0 */
{  /* drop */
}
/* stack now: ... p1 */
goto label1;
label0:
/* stack now: ... p1 p0 */
{   /* drop */
}
/* stack now: ... p1 */
label1:
```

```
/* stack now: ... p1 */
{   /* exit */
    _c_result=p1;
    return (_c_result);
}
}
```

In this example, we can see that the output code is not very clean. But with *f2c*, the C language is more essn as an intermediate languague than as a programmer one because the aim of *f2c* is not to produce a beautiful code but an easily optimizable one. With only a copy propagation optimization this code becomes clearer. In *f2c*, it's the C compiler which has to do this, but it's not always the case for every source-to-source translator.

2.4.2 Compiler

Source-to-source translators are not the only softwares which remove the stack in a stack-based language. Stack-based compilers do the same thing, but they convert this language into a low level one (it's often into mnemonic instructions). So their algorithm must be useful for us. We will only see two examples (*RAFTS*, a Forth compiler and *LaTTe* a JAVA Bytecode compiler).

2.4.2.1 RAFTS

RAFTS [6] is a framework for compiling Forth code. It tries to produce fast and efficient code, so it needs to use some optimization techniques and interprocedural register allocation to eliminate nearly all stack accesses because they slow down the execution of the program. *RAFTS* compiles all of *Forth*, including unknown stack heigths.

RAFTS uses several steps to compile *Forth* code. The first step is to split the code into basic blocks. A basic block is a set of instructions which only contains simple primitives like literals, constants, variables, `+`, `!` and stack manipulation words. So a basic branch does not contain any branch or jump : all primitives are executed sequentially. Then *RAFTS* builds a data flow graph of this basic block (see figure 2.3).

Just after that, it converts *Forth* primitives into mnemonic instructions and transforms all stack items into unlimited pseudo-registers. So all stack accesses within a basic block have been eliminated and the DAG is now an instruction DAG (see figure 2.3). Then an instruction scheduler orders the

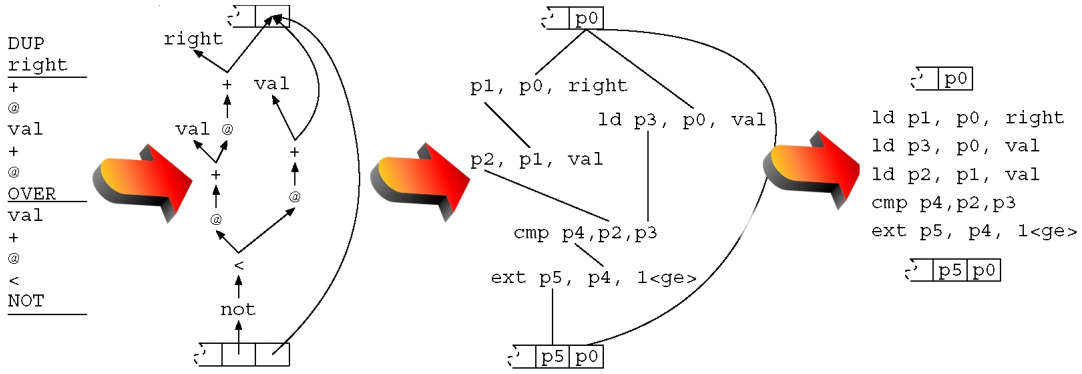

Figure 2.3: RAFTS conversion steps for a basic block

nodes of the instruction DAG, i.e. it transforms the DAG into a list. This list is optimized for reducing register dependencies between instructions.

Now, we have a set of mnemonic blocks, but we have to connect them thanks to control structures. Control flow splits (`IF`, `WHILE` and `UNTIL`) are easy to transform but control flow joins (`ENDIF` and `BEGIN`) are a little harder because the corresponding stack items of the joining basic blocks usually do not reside in the same register. So *RAFTS* needs to move some values around to have the same structure.

In order to have a faster output code, three good register allocation algorithms are proposed : Graph coloring register allocation [2], hierarchical graph coloring [3] and interprocedural allocators [4].

2.4.2.2 LaTTe

Today, we need faster and faster execution for our JAVA applications. Higher JAVA performances can be achieved by *Just-In-Time* (JIT) compilers which translate the stack-based bytecode into register-based machine code. One crucial problem in JAVA JIT compilation is how to map and allocate stack entries and local variables into registers efficiently and quickly as to improve the JAVA performance.

LaTTe [19] is a JAVA JIT compiler that performs fast and efficient register mapping and allocation for SPARC machines. *LaTTe* converts JAVA bytecode (a stack-based language) to SPARC mnemonic. It uses severals

steps for this :

- First, LaTTe identifies all control join points and subroutines in the JAVA bytecode via a depth-first traversal in order to build a control flow graph (CFG).
- Then, it converts this bytecode into a CFG of pseudo SPARC instructions with symbolic registers.
- Optionally, some traditional optimizations are performed.
- In the fourth step, LaTTe performs a fast register allocation, generating a CFG of real SPARC instructions.
- Finally, the graph is converted into a list of SPARC instructions.

In order to transform the stack into registers, LaTTe uses symbolic pseudo-SPARC registers whose names are composed of three parts :

- The first character indicates the type : `a` for an address (object reference), `i` for an integer, `f` for a float, `l` for a long and `d` for a double.
- The second character indicates the location : `s` for operande stack, `l` for local variable and `t` for temporary variable used by LaTTe.
- The remaining number distinguishes the symbolic registers.

For example, `il2` represents the second local integer register. But at the end of the algorithm, *LaTTe* transforms these pseudo-registers into real ones. To do that so as to increase performance, LaTTe uses two passes for each extended basic block :

An extended basic block is a tree region which has a single entry but multiple-exit subgraphs like tree

- **The backward sweep algorithm** is a post-order traversal which collects information on the preferred destination registers for instructions.
- **The forward sweep algorithm** is a depth-first traversal which performs the real register alloation using that information.

Sometimes, we need to move register in order to reconciling register allocation at region join points because LaTTe use these two algorithms on each extended basic block independently. So two blocks may not use the same register for the same item on the stack.

This method is very efficient : the output code of LaTTe is on average two times faster than with the SUN JIT and this speed comes particularly from the register allocation algorithm.

Chapter 3

BibTEX++

As we have seen in the section 2.1, *BibTEX* is a very good tool to produce bibliographies but it begins to become old : it was created during the 80's and the last modification was the management of 8 bit characters in 1990.

Today, the LaTeX community needs lots of new functionnalities like :

- Multilingual bibliography. It's easier and easier to obtain documents from foreign countries, so, sometimes, we have to deal with bibliography which contains references with asian or arabic characters.
- Access to bibliography databases from the Internet. Lots of them are available on the net [13] so it would be better if the tool connects automatically to an online database and recovers temporarily all useful references.
- Style programming language more expressive. Today, there are only few people that are able to create a new bibliography style or to understand an old one. We've already seen how weird is the *bst* language.
- A software which can easily evolve. In the future, we might need new features and it will be a shame to rewrite a new*BibTEX-like* tool.
- Compatible with *BibTEX*. Lots of stuff had been written for *BibTEX* like styles, bibliography databases... so it's better if this new tool can use them.

It was in this context that the project *BibTEX++* was born.

3.1 Presentation of BibTEX++

One of the most constraint for *BibTEX++* is the compatibility with *BibTEX*, so its architecture must look like *BibTEX* one. But we also need to easily change the comportement of *BibTEX++*, in particular the input and output data. So we arrive at a structure like the figure 3.1.

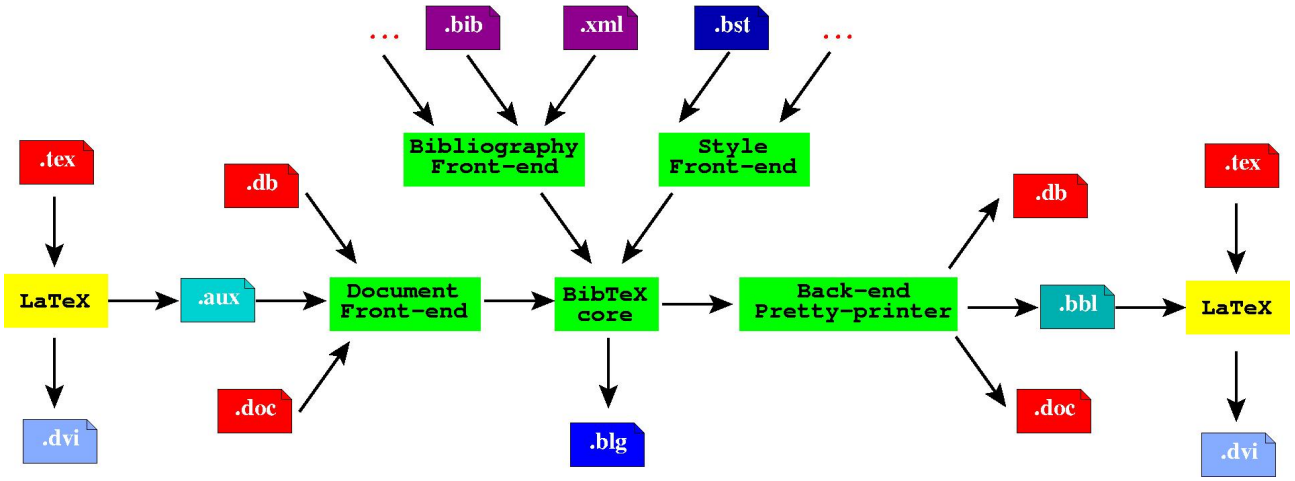

Figure 3.1: *BibTEX++* Workflow

The aim of this structure is to change all file formats with a *BibTEX++* plugin. For example, there are lots of different formats for bibliographic databases (*tib* format, XML ...), so with a plugin, we will be able to use a *.xml* instead of *.bib*. This plugin will contain an XML parser/lexer and a tool which converts the syntaxical tree of the *.xml* file into an internal *BibTEX++* database object. It will be the same thing for style files and input files, so *BibTEX++* may become a bibliographic software for LaTeX and *Micro$oft Word*.

Precisely, we need a language for the bibliography style. But there are lots of constraints :

- A well-known language (not like weird *bst* one).
- With UNICODE, network... support.
- That can deal with big programs.

➥ Portable, LaTeX is almost present on every architecture.

It is why JAVA has been choosen for both style and software language. So, to obtain compatibility between *BibTEX* and *BibTEX++*, we need to transform bst programs into JAVA ones.

3.2 BiSTrO architecture

The main part of my work during this application was to create the software which will transform a *bst* style file into a JAVA one. This software is called *BiSTrO* (BibTEX Style Translator and Optimizer). One of the aims of this is to enable *BibTEX++* users to easily write new styles or modify existant ones. So the JAVA generated styles must be easily comprehensible because they will be used as skeleton for new styles.

But all the source-to-source translators or compilers in section 2.4 try to optimize the output in order to speed up the execution, but we need here to increase clearness. So just after the transformation into JAVA, we will have to add some optimizations to increase it. But, depends of its use the *BibTEX++ compiler* will need speed or optimization.

If someone just wants to use *BibTEX++* with an old *BibTEX* style, he will never look at the generated JAVA file, but he can be impatient. In this case, we will remove these optimizations and also use a JAVA cache to stock all the JAVA Style already created by *BibTEX++*.

But if someone just wants to create a new JAVA style based on a *bst* old one, he will want to understand the style file. In this case we will have to optimize the output code and to enable users to only use the *BibTEX++ style compiler*.

3.2.1 Description

In thinking at all these constraints, we create *BiSTrO* (Bibtex Styles Translator and Optimizer) with the architecture on figure 3.2. *biSTrO* is composed by four components that we will see more in details in the next sections :

The Lexer/Parser : it transforms the .bst file into a bst Abstract Syntax Tree (AST) in order to be easily analysed. This parser had been generated by *SableCC* [7] with a home-made bst grammar.

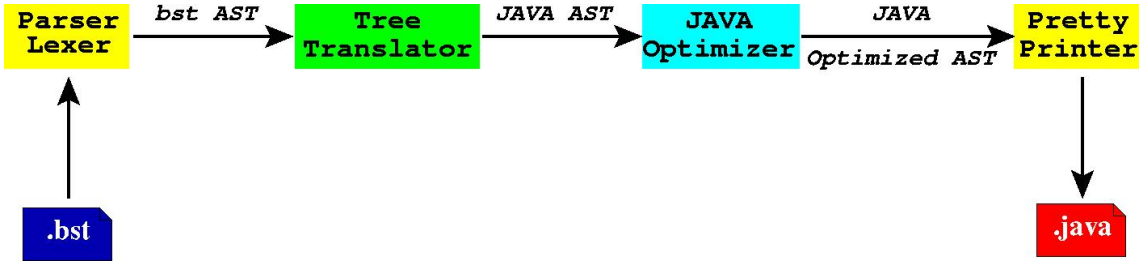

Figure 3.2: *BiSTrO* Architecture

The Tree translator : first, it makes several passes in the bst AST in order to obtain information about bst functions, elements on the stack and possibility to remove the stack. Then, it converts the bst AST into a JAVA AST.

The JAVA optimizer : it modifies the JAVA tree in order to optimize the legibility of the generated code. It uses several independent optimizations to allow the user to choose which optimizations he wants to use at the time of the execution.

The Pretty Printer : it transforms the optimized (or not) JAVA tree into a real JAVA file. It also indents this code to increase its legibility.

3.2.2 The Parser / Lexer

There are only few compiler compilers which allow to create Abstract Syntax Tree (AST). Most of them, execute an *action code* as soon as the generated parser discovers an element of the grammar (like *YACC* [11] and *JAVA CUP* [9] for example). So it's difficult to generate multiple pass compilers with these compiler compilers without rebuilding an AST. For JAVA, the most known which permits to create AST is ANTLR2.xx [15], a JAVA implementation of PCCTS [14].

Nevertheless, ANTLR lexer does not support 16 bits unicode character input. It is why we chose here the *SableCC framework* [7]. SableCC sits in the middle between front-end compiler compilers (a lexer plus a parser) and end-to-end compiler compilers which provide complete compilers. We chose it because it has lots of useful features like :

- ➥ It separates machine generated code and human written code. This simplifies the maintenance of BiSTrO.
- ➥ We don't need runtime library at the execution (unlike JAVA CUP [9]). So users of BibTEX++ only need a recent JAVA virtual machine.
- ➥ It is free and under Lesser General Public License (LGPL). The source are also availabled on SourceForge (`http://sourceforge.net/projects/sablecc/`).
- ➥ It creates at the same time the lexer and the parser.
- ➥ The parser automatically builds the AST of the compiled program and the node classes are created during the compilation of this parser.

So, to generate this lexer/Parser we only need to create the bst grammar file. SableCC also automatically provides an AST analyser to helping the development of the next step of a compiler : the analyser.

The syntax of the grammar file is really easy to understand. It contains up to six parts (all of them are optional) :

`package_declaration :` it specifies the root package for all classes generated by SableCC.

`helper_declaration :` a helper is a character set or a regular expression denoted by an identifier. When SableCC sees a helper identifier in a regular expression, it replaces it semantically by its definition.

`states_declaration :` it contains the name of each state used in the `token_declaration`.

`token_declaration :` it contains all the definitions of the token used by the lexer. We can use states (like in *GNU FLEX*) in order to restrict some use of tokens.

`ign_tokens :` it is the list of the tokens that are ignored by the parser.

`production :` it is where we declared all the grammatical rules between tokens.

You can find the complete grammar for the bst language in the appendix A, but in order to show you a real example, this is the grammar file for the *.aux* file :

```
Package auxi;

Helpers
    unicode = [0..0xffff];
    lf  = 10;
    cr  = 13;
    bs  = 92;
    eof = 0x1c;
    line_terminator = lf | cr | cr lf | eof;
    all = [unicode - [cr + lf]];
    ident_char = [unicode - ['{' + [cr + lf]]];
    info_char = [ident_char - ['}' + ',']];
    command = bs ident_char*;
States
    init, bibinfo, junk;
Tokens
    {bibinfo} comma             = ','   ;
    {bibinfo} info              = info_char+ ;
    {bibinfo} l_brace           = '{'   ;
    {bibinfo -> junk} r_brace   = '}'   ;
    {init -> bibinfo} citation  = bs  'citation' ;
    {init -> bibinfo} bibdata   = bs  'bibdata' ;
    {init -> bibinfo} bibstyle  = bs  'bibstyle' ;
    {init -> bibinfo} biblang   = bs  'select@language' ;
    {init -> junk} ident        = command;
    {junk -> init} forget       = all* line_terminator+;
Ignored Tokens
    forget,ident;
Productions
    expr_list = expr+;
    expr =
         {citation} citation l_brace info r_brace|
         {bibdata} bibdata l_brace args r_brace|
         {bibstyle} bibstyle l_brace info r_brace|
         {biblang} biblang l_brace args r_brace;
    args = info args_end*;
    args_end = comma info;
```

3.2.3 The Tree translator

The advantage of having an AST at the exit of the parser is the possibility to analyse the code with several passes. In order to make this easier, SableCC creates, when you compile the parser/lexer, several analysers for your specific AST. When you want to create your own one, you only have to extend yours from an existent one and change the behavior of some functions. There are three functions per node : one which is executed before entering in a node, one after and one inside.

The aim here is to transform this bst tree into a JAVA one. So we need information to do that, it is why we make several passes. So we use several analysers to do this :

`ParseFunc.java :` It searchs information about functions in the bst tree. For each function, we try to obtain the name of the function, the number of input and output parameters and the possibility of removing the stack in this function : it is not always possible (see section 3.3.6).

`ParseVar.java :` It tries to find the type of the arguments of all functions. When it can not find this type, it indicates that we will have to use a polymorphic *Cell* object which can be either a string or an integer.

`ParseDeclaration.java :` It analyses the body of the function and the stack to find how many JAVA local variables we will have to use and what is their type.

`OutFunc.java :` With all this information, it transforms into JAVA functions where we can remove the stack with local variables instead of stack items.

`OutStack.java :` It transforms into JAVA functions where we can't remove the stack. But the bst stack is replaced by a JAVA stack.

So just after these four passes, we have a JAVA abstract syntax tree, but this code is not optimized at all, so we need to clarify it a little. It is why we use an optimizer.

3.2.4 The Optimizer

We decided to use several types of optimizations which are independent in order to provide a final optimization which is fully customable by the

user. But we do not need very complex optimizations because the aim of this is to increase the legibility and not speed the execution. Another reason for using simple optimizations is that the input code was written by human programmers in a not very understandable language, so they tried to write this code properly.

We just use eight different functions in order to do that :

An `if` optimizer : it just removes all the empty `then` or `else` blocks.

A copy and constant propagation : we can see in the example section 3.3.2 page 24 that the non-optimized output code contains lots of useless variables because there is a variable for each element on the stack. It will increase the quality of the next dead code elimination.

A dead code elimination : associate with the propagation optimization, it removes lots of useless definitions, because it deletes all *write after write* dependencies.

A boolean translator : in bst language, we don't have any boolean type, so this optimization tries to transform integer into boolean in `if` and `while` conditions. For exemple, it will transform `if( BuiltIn.equal( i1 , i2 ) > 0 )` to `if( i1 == i2 )`.

Some other small optimizations : like peep-hole optimization and poor-man partial evaluation : transforming `a1 = a0 + 0;` into `a1 = a0;` or ``` ``some''+``thing'' ``` into ``` ``something'' ```... These optimizations only try to increase the legibility on the JAVA code.

3.3 Translation

With all these steps, we succeed in transforming *bst* language into JAVA, but we will see in the section what are the convention we use in this transformation and the difficulties we met.

3.3.1 Stack representation

In all functions without unknown stack depth, we transform the stack in JAVA local variables in order to simplify the notation and increase the clarity of the output code. In *bst* language there are two different types of items on

the stack : *string* and *integer* which are implicit. Outside of the stack there are lots of other variables (see section 2.3), but we can use for them their name in the bst language. However the elements on the stack don't have any name, we have to find one in JAVA.

Thus in JAVA, we use three types of local variables : `String`, `int` and `Cell` which is a polymorphic object (it can be a string or an integer depending of the context). We use this `Cell` when *BiSTrO* fails to find the original type of an element.

The name of a local variable is composed of two parts : the first letter indicates the type of the variable (`s` for `String`, `i` for `int` and `c` for `Cell`). The second part is a number which indicates the height of the element in the local stack (the deepest element of the function uses the number 0). The input paramaters use the same notations (see figure 3.3).

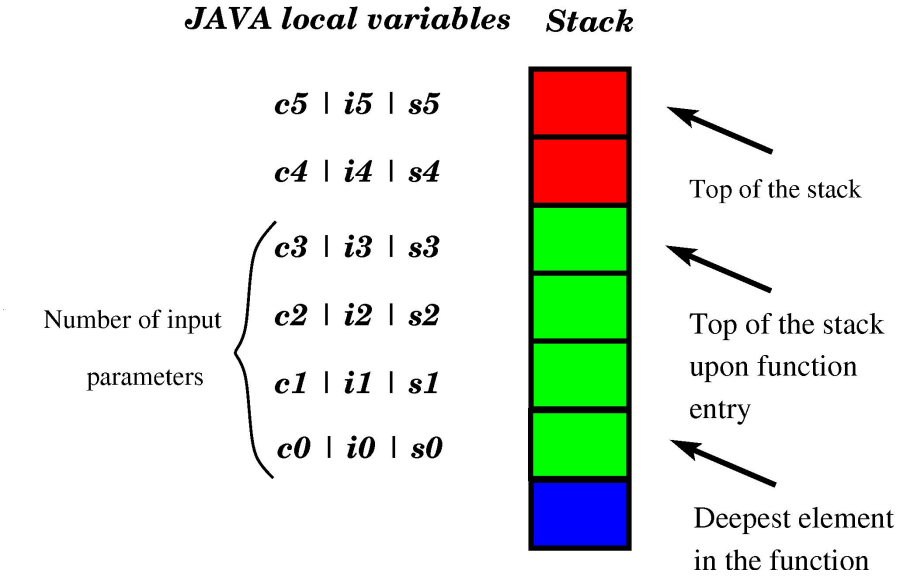
JAVA local variables
Stack
c5 | i5 | s5
c4 | i4 | s4
c3 | i3 | s3
c2 | i2 | s2
c1 | i1 | s1
c0 | i0 | s0
Number of input parameters
Top of the stack
Top of the stack upon function entry
Deepest element in the function

Figure 3.3: Variable notations in *BiSTrO*

3.3.2 Built-in functions

We have already seen that we can use 37 built-in functions in *BibTEX* for writing new styles. Some of these functions can easily be transformed into

JAVA ones (+ for example), but it is more difficult for other functions (like `format.name$` which formats a name inside a name list). So, for all these functions, we create a class which contains the JAVA code of them. The generated code only calls these functions.

For example, the translation of this bst function :

```
FUNCTION {format.lastchecked}
{ lastchecked empty$
    { "" }
    { inbrackets "cited " lastchecked * }
  if$
}
```

will be (without optimizations)

```
public String format_lastchecked( )
{
   String s0 , s1;
   int i0;
   s0 = lastchecked;
   i0 = BuiltIn.empty( s0 );
   if( i0 > 0 )
   {
      s0 = "";
   }
   else
   {
      inbrackets( );
      s0 = "cited ";
      s1 = lastchecked;
      s0 = s0 + s1;
   }
   return( s0 );
}
```

We can see in this small example that *BiSTrO* can transform a bst function in a JAVA one. Some built-in functions (like `empty$`) are converted into calls for home-made function (here `BuiltIn.empty`) and other (like `*` which concatenates two strings) are directly converted in JAVA (in `+`). We can also

note here the usefulness of the optimizations. Indeed, this JAVA function is not very easy to understand but if we use our optimizations, the output will be something more understandable :

```
public String format_lastchecked( )
{
   String s0;
   if( BuiltIn.empty( lastchecked ) > 0 )
   {
      s0 = "";
   }
   else
   {
      inbrackets( );
      s0 = "cited " + lastchecked;
   }
   return( s0 );
}
```

3.3.3 Type inference on stack item

In the previous example, we can see that the type of all stack elements and return parameters were found. It is easy to find their type because they really represent items on the stack. There are only four ways to put an element on the stack in bst language :

- Write its value, like ``cited'' or #21, and we know immediately the type of this object.
- Write the name of a global variable or field, but when you declare these elements, you have to indicate their type, so it is not a problem.
- Use a built-in function, but *BiSTrO* knows the return parameter type of all these functions.
- Use a home-made function, and you have just to iterate the algorithm to find its return type.

Therefore the type of all these object are not very difficult to find : *BiSTrO* find them in only one pass, because in bst language, a function

have to be declared before being used. But it is more difficult to find the type of the argument of a bst function because, on the one hand, we don't know (in the function) where these elements were put on the stack (so we can't use the previous method) on the other hand, some function can take enter variables which can be of different types. For example, in bst language, we can create a function which takes two parameters. If the first argument is the string `''integer''` then the second argument is an integer and we add 5 to it, else we return the number of letters of the second argument (which is a string). So, sometimes we will not be able to find the type of each parameter and we will have to use the `Cell` polymorphic object.

For these reasons, *BiSTrO* use an algorithm which is quite similar to the first one, but in the reverse order. We use the entry parameters of functions (built-in or home-made) to find the type of the arguments. For example, if a function takes only one argument and the first primitive is `write$`, *BiSTrO* will deduce that this argument is a string. But there are some functions which modify the stack (`duplicate$`, `pop$` and `swap$`), so *BiSTrO* uses some abstractions to indicate where are the entry parameters on the stack (see figure 3.4)

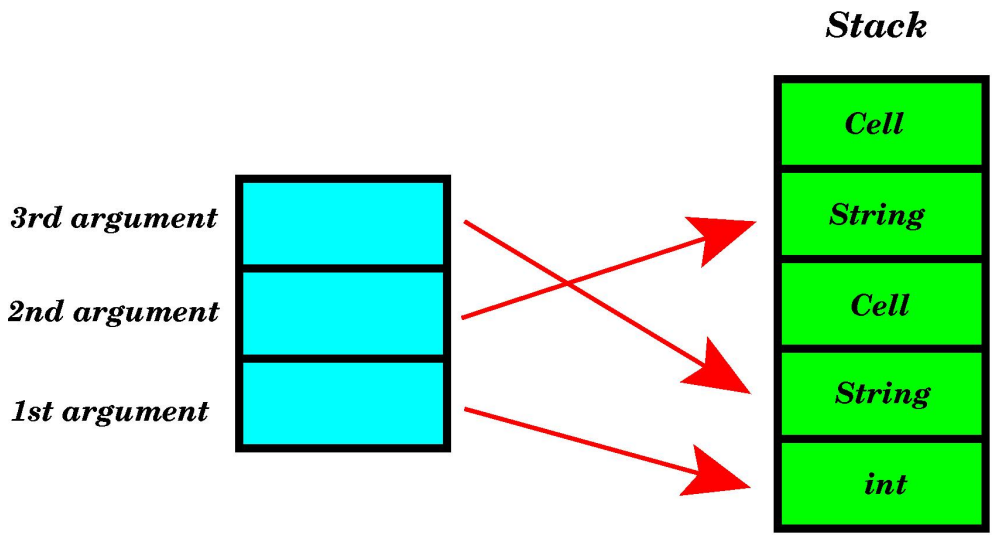

Figure 3.4: Finding input types

3.3.4 Return parameters

In bst language, you can put how many elements you want in the stack, so there are functions that return more than one element in the stack. But it is not possible to do it in JAVA (functions can only return zero or one variable). To correct it, *BiSTrO* puts all these variables in a `Vector`, and when another function calls this one, we just have to extract the variables from the `Vector`.

3.3.5 Control structures

In bst language, there are only two types of contol structures : `if$` and `while$`. So *BiSTrO* just need to convert them into JAVA `while` and `if` structure. But sometimes, you do not have the same stack state if you execute a branch or the other inside an `if$`. Just after this condition structure, the state of the stack is unknown (we do not know the stack depth or the type of an element without some very advanced abstract interpretation), consequently we can not use traditionnal transformations in this function. So we will see in section 3.3.6 how we transform these functions.

Another problem of the same type is when inside a `while$` block, you put (or remove) an element in the stack. Depending on how many times this block will be executed (what is unknown at compilation time), the stack depth will be different. So we also need to use transformation technique for unknown stack depth.

These cases are quite unusual but there is a function (which is called *format.names*) that contains this type of weird code (see appendix B). However near 95% of BibTEX styles use this function, so, in order to increase the ligibility of the output code, we decide to perform some pattern matching in order to transform this function in a bst one that perform the same thing but without unknown stack depth.

3.3.6 Unknown stack depth

In the previous section, we saw that there are some functions in which we can know everywhere the stack depth. So, in order to provide a fully BibTEX compatible software, we need to transform this type of function into JAVA one. To do that we decide to use JAVA stack, so we provide to the style an interface to a stack with all the functions that can be usefull (`pop`, `push`, `swap` and `dup`). The output will be something like that :

```
public void format_bdate( )
{
   stack.push( year );
   stack.push( BuiltIn.empty( stack.pop( ).getString( ) ) );
   if( stack.pop( ).getInt( ) > 0 )
   {
      stack.push( "there's no year in " );
      stack.push( BuiltIn.cite( bib ) );
      stack.push( stack.pop( ).getString( ) + stack.pop( ).getString( ) );
      System.err.println( "Warning : " + stack.pop( ).getString( ) + "." );
   }
   else
   {
      stack.push( year );
   }
}
```

So, in a java style file, there can be both functions without a stack and functions with a stack. But if a function calls another one which uses a stack, the function have to have a stack. So the stack propagates itself fast.

3.4 Optimizations

In this section, we will see more in details two optimizations used in *BiSTrO*.

3.4.1 Copy propagation

A copy statement is of the form `f=g;`. In other words, a single variable is being assigned the value of another variable. No other statements are copies.

The Dragon Book[1] says this about copy propagation:
"The idea behind the copy-propagation transformation is to use `g` for `f` wherever possible after the copy statement `f=g;`." In other words, whenever `f` appears on the right hand side of an assignment, we put `g` in it's place. We can do this so long as `g` and `f` do not get assigned anywhere else (in compiler language that assignment is known as a "kill"). Kills always take place after the statement. So, if given the statements:

```
f=g;
```

```
f=f * 2;
```

We can replace `f` with `g` on the right hand side of statement 2 to get `f=g * 2;`. Associated with a *dead code elimination*, it can reduces a lot the number of variables and the size of the program.

3.4.2 Constant propagation

Constant propagation looks like *copy propagation*, but here, we do not propagate variables but constants, so the next code :

```
f=5;
f=f * 2;
```

will be transform into :

```
f=5;
f=5 * 2;
```

It also have the same qualities and uses than the *copy propagation*. In *BiSTrO*, we do constant and copy propagation in the same pass because their code are very similars.

3.4.3 Dead code elimination

Dead code elimination is the ability of the compiler to remove statements that are known as *dead*. A statement is dead if it computes values that are never going to be used and has no side-effect. For our purposes, dead code can only be copies whose left hand side is not used again (ie, because of copy propagation, the copy is now dead). For exemple in the previous exemple, the first statement at the output of the optimization is dead because `f` is re-assigned in the next statement and was not used before. So the optimized code of :

```
f=g;
f=f * 2;
```

will be :

```
f=g * 2;
```

3.5 Security

Today, we can't write a software without thinking at the security and viruses because there are everywhere and also inside *VBscript* macros in *Micro$oft Word* documents. So, we can imagine that viruses may be inside *BibTEX++* styles. They are written in JAVA so they can access to every files, the network... *BibTEX++* styles create a *.bbl* file which contains the bibliography, but this file is written in LATEX code so we can insert malicious code in it (the TEX command `write18` can be configured to execute shell commands on the machine).

In order to limit the creation of styles containing viruses or malicious code, *BibTEX++* uses the *JAVA security manager*. It's a feature of JAVA which limits some operations (like file and network accesses) to classes.

So, by default, we forbid to all styles to do something dangerous (file reading and writing, accessing to environnement variables, accessing to the internet ...). If a style need one of this access, the user will have to modify the `policy` file which contains the security rules. At the installation, this file is :

```
grant {
permission java.util.PropertyPermission "bib_max", "read,write";
};

grant codeBase "file:${bib_lib}" {
permission java.io.FilePermission "<<ALL FILES>>", "read,write,execute";
permission java.util.PropertyPermission "bib_cache", "read";
permission java.util.PropertyPermission "bib_lib", "read";
permission java.util.PropertyPermission "java.class.path", "read";
};
```

All classes but those inside the *BibTEX++* library, (so only plugins and styles) can only access one environnement variable : `bib_max`. It is the maximum length of a string in a style.

Chapter 4

Results

The *BibTEX++* program is composed of two executables :

`bibtex++` which creates a `.bbl` file for LaTeX from `.aux` file (indicating the used references, the style and the reference database), a `.bib` which is the bibliography database and the style (in JAVA or bst language). the syntax is : `bibtex++ [options] filename (without extension)` and the options are :

- `-v` : print version information and exit
- `-O` : optimize the java style (if you generate one)
- `-h` : print help text
- `-n` : create a pretty.bib

`BiSTrO` which convert a `.bst` style file into a JAVA one. The syntax looks like the `bibtex++` one : `BiSTrO [options] filename.bst` where the options are :

- `-v` : print version information and exit
- `-O` : optimize the java style (if you generate one)
- `-h` : print help text

4.1 Speed and execution time

All these tests were made on a *Athlon XP 2000* with 512 Mo of RAM with *j2re* (Java 2 Runtime Environment) version 1.4.1 for linux. The *BibTEX++*

programs were compiled by the sun `javac` with the option -O (optimizes the code).

In order to test the compatibility of *BiSTrO* with *BibTEX* styles, we compiled all the 152 styles of the *MikTEX* distribution. It allowed us to find some bugs in our software and other in the old *BibTEX* styles.

It also allowed us to compare the execution time and the size of the styles between the optimized version of *BiSTrO* and *BibTEX++* and the normal one.

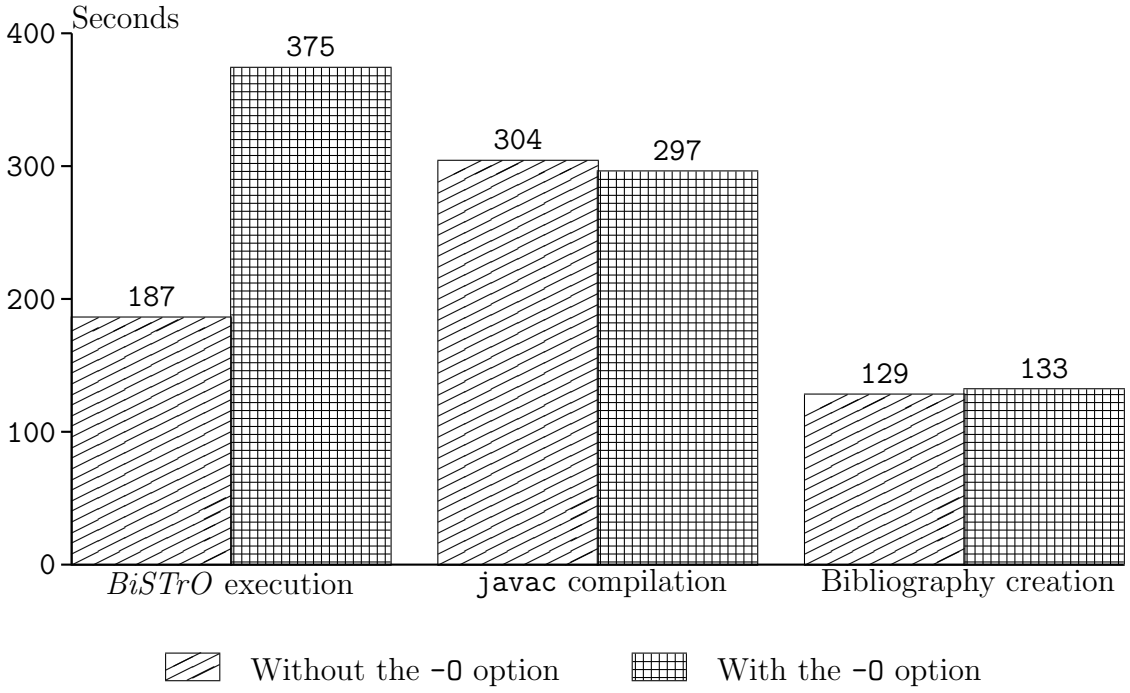

Figure 4.1: Execution time of some *BibTEX++* programs on 152 styles

In figure 4.1, we can see that it is two times slower to optimize the JAVA style file. Nevertheless, the total compilation of a bst style file to a class file is quite fast : only 3.2 seconds without optimizations and 4.4 seconds with all of them.

The execution of *BibTEX++* is far slower than the original *BibTEX* (near 0.04 second on a small example), but this is not disturbing for a normal user because he just has to compile once the *.bst* file (it takes 3.2 seconds) and the other times the creation of the `.bbl` file will only take 0.8 second.

Finally, the optimizations are not very useful for a standard *BibTEX++* user : it increases the compilation time but does not decrease the execution time. Nevertheless, we will see in the next section that they are useful for style designers.

4.2 BibTEX++ styles and optimizations

We have already seen that there are two ways for creating a new style : modify an existent one or create a new one from scratch. For both, it is easier with *BibTEX++* than with *BibTEX* because the JAVA language is more expressive, higher level and comprehensible than the bst language.

In order to help developpers who want to reuse an old style, they can optimize the output code of *BiSTrO*. Nevertheless, as we just see, these optimizations do not decrease the execution time of the style (because `javac` also optimizes the code when we compile it). So these optimizations are just useful for a style designer.

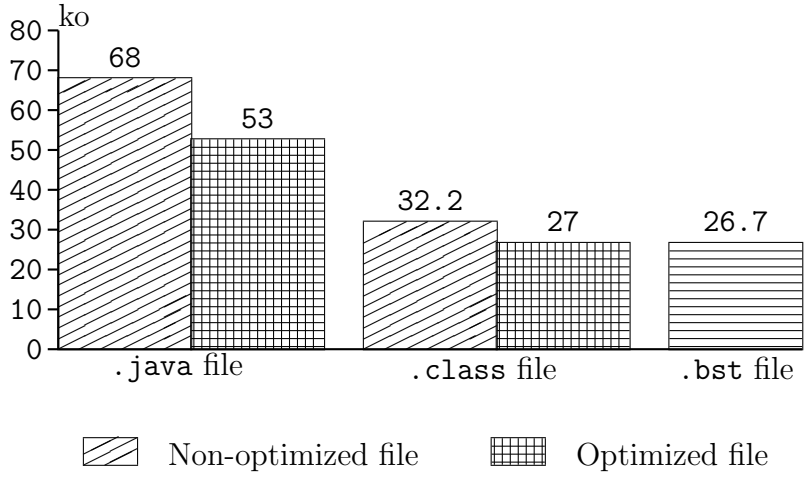

Figure 4.2: Mean size of a some style formats

We can see in the histogram in the figure 4.2 that these optimizations decrease the size of the `.java` style file by 22% and of the `.class` file by 16%. So they reduce the length of the code (it is often easier to understand a smaller code) but also increase its legibility.

For example, the following function is not easy to understand and is a bit long :

```
public void new_sentence( )
{
   int i0 , i1;
   i0 = output_state;
   i1 = after_block;
   i0 = BuiltIn.equal( i1 , i0 );
   if( i0 > 0 )
   {
   }
   else
   {
      i0 = output_state;
      i1 = before_all;
      i0 = BuiltIn.equal( i1 , i0 );
      if( i0 > 0 )
      {
      }
      else
      {
         i0 = after_sentence;
         output_state = i0;
      }
   }
}
```

But the optimizations will transform it into a more understandable function :

```
public void new_sentence( )
{
   if( output_state != after_block )
   {
      if( output_state != before_all )
      {
         output_state = after_sentence;
      }
   }
}
```

We can see in this example, lots of different optimizations that we use :

- All empty ***then*** *blocks* had been removed.
- We propagated all the global function. the `i0 = after_sentence; output_state = i0;` have been transform into `output_state = after_sentence;`. We do not use any local variable.

➥ We convert some built-in functions to boolean. The `BuiltIn.equal( i1 , i0 ) > 0` have been transform into `i1 > i0`.

So we can see here that all these transformations are pretty efficient. Indeed, the optimized function is much more readable that non-optimized one.

In figure 4.2, we can also note that the optimized `.class` file has almost the same size than the original `.bst` file. But this class file was automatically generated, so we can imagine that hand-made *BibTEX++* style files will be smaller than *BibTEX* ones, so they will be easily downloadable and sharable.

Conclusion

In this report, we introduce you a new bibliographic software : *BibTEX++* which might become the successor of *BibTEX*.

In order to provide a fully *compatible-with-BibTEX* software, we developed a system which converts the *BibTEX* style files (written in bst language) to *BibTEX++* ones (in JAVA). This system is called *BiSTrO* (BibTEX Style Translator and Optimizer).

We have seen in section 3.3 how we transform a bst language which is stack-based into java code. These transformation rules are based on the *f2c* work [5], but have been adapted for the bst and JAVA languages.

Nevertheless, the output code of this translator is hard to understand for designers. Indeed they sometimes modify existent styles in order to create their own ones. So the *BiSTrO* output code must be more legible. It is why we implement different optimizations in this translator, like copy and constant propagation, dead code elimination, ...

BiSTrO had been tested on more than 150 *BibTEX* styles and gives clear JAVA styles when used with `-O` option. Today, *BibTEX++* is fully operating (it generated the bibliography of this report) and is available at `https://picolibre.enst-bretagne.fr`.

Nevertheless, lots of things still have to be done :

- We have already spoken about the plugins in section 3.1, but *BibTEX++* does not manage them at all yet. So we will have to change that.
- We need to develop *BibTEX++* styles in Java in order to show all the possibilities of this software (like Internet accessing, ...).
- Once the plugins have been managed, we will have to create some, in order to provide new features like Unicode bibliography in LaTeX and possibility of creating bibliographies in Micro$oft Word documents.

- *BibTEX++* is quite slow (near 20 times slower than *BibTEX*). We may increase its speed if ever need. But we've never seen a real usage where spending few seconds on BibTEXing was an issue yet.

Appendices

Appendix A

bst grammar

```
Package bst;

Helpers

  cr = 13;
  lf = 10;
  tab = 9;
  eol = cr lf|cr|lf;
  all=[0..0xFFFF];
  not_cr_lf=[all-[cr+lf]];
  lowercase=['a'..'z'];
  uppercase=['A'..'Z'];
  digit=['0'..'9'];
  letter=lowercase|uppercase|'$'|'.'|'-'|'>'|'<'|'+'|'&'|'/';
  letint=letter|digit;

Tokens

  entry='ENTRY';
  execute='EXECUTE';
  function='FUNCTION';
  integers='INTEGERS';
  iterate='ITERATE';
  macro='MACRO';
  read='READ';
  reverse='REVERSE';
  sort='SORT';
  strings='STRINGS';

  greater='>';
```

```
lesser='<';
equal='=';
plus='+';
minus='-';
concat='*';
assign=':=';
l_brace='{';
r_brace='}';
simple_quote='';

add_period='add.period$';
call_type='call.type$';
change_case='change.case$';
chr_to_int='chr.to.int$';
cite='cite$';
duplicate='duplicate$';
empty='empty$';
format_name='format.name$';
if='if$';
int_to_chr='int.to.chr$';
int_to_str='int.to.str$';
missing='missing$';
newline='newline$';
num_names='num.names$';
pop='pop$';
preamble='preamble$';
purify='purify$';
quote='quote$';
skip='skip$';
stack='stack$';
substring='substring$';
swap='swap$';
text_length='text.length$';
text_prefix='text.prefix$';
top='top$';
type='type$';
warning='warning$';
while='while$';
width='width$';
write='write$';

integer='#' '-'? digit+;
text=letint+;
text_int=letint+;
string='"' [not_cr_lf - '"']* '"';
```

```
  blank=(' '|tab|eol)+;
  comment='%' not_cr_lf* eol;

Ignored Tokens

  comment,
  blank;

Productions

  command_list = command+;

  command =
    {entry_cmd} entry_cmd |
    {execute_cmd} execute_cmd |
    {function_cmd} function_cmd |
    {integers_cmd} integers_cmd |
    {iterate_cmd} iterate_cmd |
    {macro_cmd} macro_cmd |
    {read_cmd} read_cmd |
    {reverse_cmd} reverse_cmd |
    {sort_cmd} sort_cmd |
    {strings_cmd} strings_cmd;

  /* Commands */

  entry_cmd =
    entry
    entry_field_list
    entry_integer_list
    entry_string_list;

  execute_cmd =
    {function} execute l_brace [funcname]:text r_brace |
    {built_in} execute l_brace [builtinfunc]:built_in r_brace;

  function_cmd =
    function l_brace [name]:text r_brace expr_list;

  integers_cmd =
    integers integers_list;

  iterate_cmd =
    {function} iterate l_brace [funcname]:text r_brace |
    {built_in} iterate l_brace [builtinfunc]:built_in r_brace;
```

```
macro_cmd =
  {macro_text} macro [ll]:l_brace [name]:text [lr]:r_brace [rl]:l_brace
               [def]:string [rr]:r_brace |
  {macro_int} macro [ll]:l_brace [name]:text_int [lr]:r_brace [rl]:l_brace
               [def]:string [rr]:r_brace ;

read_cmd =
  read;

reverse_cmd =
  {function} reverse l_brace [funcname]:text r_brace |
  {built_in} reverse l_brace [builtinfunc]:built_in r_brace;

sort_cmd =
  sort;

strings_cmd =
  strings strings_list;

/* ENTRY command */

entry_field_list =
  l_brace entry_field* r_brace;

entry_field = [name]:text;


entry_integer_list =
  l_brace entry_integer* r_brace;

entry_integer = [name]:text;

entry_string_list =
  l_brace entry_string* r_brace;

entry_string = [name]:text;

/* INTEGERS command */

integers_list =
  l_brace integer_element* r_brace;

integer_element = [name]:text;
```

```
/* STRINGS command */

strings_list =
  l_brace string_element* r_brace;

string_element = [name]:text;

/* FUNCTION command */

expr_list =
  l_brace expr* r_brace;

expr =
  {if}        [then]:block [else]:block if |
  {while}     [condition]:block [do]:block while |
  {literal}   [lit]:literal ;

block =
  {withbraces}    l_brace expr* r_brace |
  {exec}          exec;

exec =
  {ref_func}  simple_quote [s]:text |
  {ref_built} simple_quote [builtinfunc]:built_in ;

literal =
  {integer}   [i]:integer |
  {string}    [s]:string |
  {variable}  [v]:text |
  {built_in}  [builtinfunc]:built_in ;

built_in =
  {greater}     greater |
  {lesser}      lesser |
  {equal}       equal |
  {plus}        plus |
  {minus}       minus |
  {concat}      concat |
  {assign}      simple_quote [field_var]:text assign |
  {add_period}  add_period |
  {call_type}   call_type |
  {change_case} change_case |
  {chr_to_int}  chr_to_int |
  {cite}        cite |
  {duplicate}   duplicate |
```

```
{empty}       empty |
{format_name} format_name |
{int_to_chr}  int_to_chr |
{int_to_str}  int_to_str |
{missing}     missing |
{newline}     newline |
{num_names}   num_names |
{pop}         pop |
{preamble}    preamble |
{purify}      purify |
{quote}       quote |
{skip}        skip |
{stack}       stack |
{substring}   substring |
{swap}        swap |
{text_length} text_length |
{text_prefix} text_prefix |
{top}         top |
{type}        type |
{warning}     warning |
{width}       width |
{write}       write ;
```

Appendix B

Error in format.names

This is a common format.names function of a .bst file :

```
FUNCTION {format.names} {
  's :=
  #1 'nameptr :=
  s num.names$ 'numnames :=
  numnames 'namesleft :=
    { namesleft #0 > }
    { s nameptr "{ff~}{vv~}{ll}{, jj}" format.name$ 't :=
      nameptr #1 >
        { namesleft #1 >
            { ", " * t * }
            { numnames #2 >
                { "," * }
                'skip$
              if$
              t "others" =
                { " et~al." * }
                { " and " * t * }
              if$
            }
          if$
        }
        't
      if$
      nameptr #1 + 'nameptr :=
      namesleft #1 - 'namesleft :=
    }
  while$
}
```

If you look carefully this code, you can see that the *then block* of the first `if$` inside the `while$` modifies the top of the stack (which is a string) but does not modify the height of the stack whereas, the *else block* put an element on it.

So we have used the standard conversion into JAVA (with variables instead of the stack). But this function is present in near old style, so it is a shame to have to use a JAVA stack inside all these styles.

But, we found a special transformation which allows us to not use a JAVA stack without modifying the comportment of this function. The new code of the `format.names` function with this transformation will be :

```
FUNCTION {format.names} {
  's :=
  #1 'nameptr :=
  s num.names$ 'numnames :=
  numnames 'namesleft :=
  s nameptr "{ff~}{vv~}{ll}{, jj}" format.name$
    { namesleft #0 > }
    { s nameptr "{ff~}{vv~}{ll}{, jj}" format.name$ 't :=
      nameptr #1 >
        { namesleft #1 >
            { ", " * t * }
            { numnames #2 >
                { "," * }
                'skip$
              if$
              t "others" =
                { " et~al." * }
                { " and " * t * }
              if$
            }
          if$
        }
        'skip$
      if$
      nameptr #1 + 'nameptr :=
      namesleft #1 - 'namesleft :=
    }
  while$
}
```

This transformation just gets the `'t` out of the `while$` block and transforms it into its real value (here : `s nameptr "ff vv ll, jj" format.name$`).

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] Preston Briggs. Register allocation via graph coloring. Technical Report TR92-183, 24, 1998.

[3] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *SIGPLAN 91 : Conference on Programming Language Design and Implementation*, pages 192–203, 1991.

[4] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *SIGPLAN '88 : Conference on Programming Language Design and Implementation*, pages 45–58, 1988.

[5] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, 1996.

[6] M. Anton Ertl. A new approach to forth native code generation. In *EuroForth '92*, pages 73–78.

[7] Étienne Gagnon. *SableCC, an object-oriented compiler framework*. PhD thesis, School of Computer Science, McGill University, Montreal, 1998.

[8] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeXCompanion*. Addison-Weslay, 1994.

[9] Scott E. Hudson. *CUP User's Manual*. Georgia Institute of Technology, March 1996.

[10] ISO. *Référence bibliographiques : Éléments essentiels*, 1958.

[11] Steven C. Johnson. Yacc: yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[12] Larousse. *Le Petit Larousse Illustré*, volume 1. 2002.

[13] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Citeseer, The nec research institute scientific literature digital library [online, cited Novembre 2002]. Available from World Wide Web: `http://citeseer.nj.nec.com`.

[14] Terence Parr. *Language Translation Using PCCTS & C++*. 1997.

[15] Terence Parr. Antlr, Another tool for language recognition [online]. Available from World Wide Web: `htpp://www.antlr.org`.

[16] Oren Patashnik. *BibTeXing*, 1988.

[17] Todd A. Proebsting and Scott A. Watterson. krakatoa : decompilation in java (does bytecode reveal source ?). In *Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS)*, pages 185–197, 1997.

[18] H.-P. V. Vliet. Mocha, java bytecode decompiler. Available from World Wide Web: `http://www.brouhaha.com/~eric/computers/mocha.html`.

[19] Byung-Sun Yang, Soo-Mook Moon, and Erik R. Altman. Latte: a java vm just-in-time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques,*, pages 128–138, 1999.